Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information and Computation 199 (2005) 228–252
Information and Computation
www.elsevier.com/locate/ic

Efficient instance retrieval with standard and relational path indexing ☆

Alexandre Riazanov, Andrei Voronkov *

Department of Computer Science, University of Manchester, UK

Received 1 December 2003; revised 2 September 2004
Available online 25 May 2005

Abstract

Backward demodulation is a simplification technique used in saturation-based theorem proving with superposition and ordered paramodulation. It requires instance retrieval, i.e., search for instances of some term in a typically large set of terms. Path indexing is a family of indexing techniques that can be used to solve this problem efficiently. We propose a number of powerful optimisations to standard path indexing. We also describe a novel framework that combines path indexing with relational joins. The main advantage of the proposed scheme is flexibility, which we illustrate by sketching how to adapt the scheme to instance retrieval modulo commutativity and backward subsumption on multi-literal clauses.

Keywords: Backward demodulation; Instance retrieval; Path indexing

1. Introduction

Effectiveness of *saturation-based theorem proving* in first-order logic with equality strongly depends on the use of various redundancy detection and simplification techniques. In provers implementing the superposition or ordered paramodulation calculi (see, e.g. [9]) the most widely used simplification technique is probably *demodulation*, also known as *term rewriting*. If we have derived

☆ This work was partially supported by grants from EPSRC.
* Corresponding author. Fax: +44 161 2756236.
E-mail addresses: riazanov@cs.man.ac.uk (A. Riazanov), voronkov@cs.man.ac.uk (A. Voronkov).

0890-5401/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2004.10.012

a unit equality $s \simeq t$, or it is one of the input clauses, we can simplify other clauses in the following way. A clause $C[s\theta]$, where θ is some substitution, can be replaced by a "simpler" clause $C[t\theta]$, if $s\theta \succ t\theta$, where $\succ$ is the used simplification ordering on terms.

Typically, demodulation is applied in two modes. *Forward demodulation* rewrites newly derived clauses by unit equalities that are already in the current clause set. This article deals with *backward demodulation*, which uses newly derived positive unit equalities to rewrite some old clauses. The main ingredient of any implementation of backward demodulation is a procedure for finding out which parts of which clauses can be rewritten by a given equality. This task can be usually reduced to solving the *instance retrieval* problem, i.e., finding all or some instances of a query term in a set of terms. Apart from backward demodulation in theorem proving, instance retrieval can also be used as a component of *completion* in *term rewriting* (see, e.g. [1]).

Efficient search in large term sets usually requires some kind of *indexing* (see, e.g. [3,16]). *Path indexing* introduced in [17,12] is an indexing technique suitable for instance retrieval. In this article we describe and assess experimentally a number of powerful optimisations to the standard implementation of path indexing for instance retrieval. Also, we describe a novel technique that combines path indexing with relational joins in order to provide perfect filtering for instance retrieval. The main advantage of the proposed framework is flexibility, which we demonstrate by adapting it to more complex problems, such as instance retrieval modulo commutativity and backward subsumption on multi-literal clauses.

2. Preliminaries

2.1. Instance retrieval

The work described in this article is aimed at finding efficient solutions to the following problem. Given a set of first-order terms $\mathcal{I}$ and a *query term* q, find all such substitutions θ that the *instance* $q\theta$ of q is a member of $\mathcal{I}$. We assume that $\mathcal{I}$ can be dynamically updated between calls to retrieval by adding or removing some terms. We are interested in hard cases when the *retrieval* operation is performed frequently and/or $\mathcal{I}$ grows very large. For such instances straightforward enumeration-based search is hopelessly inefficient.

It is important to emphasise that upon retrieval we are interested in finding the substitutions θ *explicitly*, in addition to the pointers to the datastructures representing the found instances $q\theta$. In the context of backward demodulation from a unit equality $s \simeq t$ into a clause $C[s\theta]$, the substitution θ is needed for constructing $C[t\theta]$. Additionally, if we allow demodulation by equalities that are not pre-ordered, the ordering constraint associated with an equality has to be checked on the retrieved substitutions. For example, if $s \not\succ t$, we have to check $s\theta \succ t\theta$.

The requirement of finding substitutions explicitly does not exclude the possibility to extract the substitution by one-to-one matching once an instance is retrieved, but in experiments the time spent on this should be included in the retrieval time to provide fair comparison of techniques.

2.2. Term indexing

The mainstream approach to implementing search in large term sets according to some specified retrieval condition is based on *term indexing* (see [3,16]). Devising a term indexing technique for

instance retrieval amounts to finding a datastructure for representing the term sets $\mathcal{I}$, called an *index*. The main purpose of the index is to facilitate fast algorithms for retrieval of instances of various query terms q. Since the database of terms is dynamically changing, we also have to provide efficient procedures for *maintenance* of the index, i.e., insertion and deletion of terms.

In what follows, we only consider indexing techniques that provide *filters* for instance retrieval, i.e., they always find (a superset of) all solutions. Moreover, we are only interested in *perfect filtering*, where all the retrieved terms are solutions.

We will shortly describe two classical examples of indexing techniques for instance retrieval: *discrimination trees* and *path indexing*. The former will be used as a benchmark in our experiments. The latter provides a basis for the main material of this article. In Sections 4–6, we describe a number of optimisations which allow very efficient implementation of standard path indexing. In Section 7, we describe a novel technique based on path indexing that extends well to other important problems: instance retrieval modulo commutativity and backward subsumption on multi-literal clauses.

2.3. Discrimination trees

A *perfect discrimination tree* [8] for a set of terms $\mathcal{I}$ can be viewed as a tree whose leaves contain terms from $\mathcal{I}$ and inner nodes are labeled by term symbols, i.e., function symbols and variables. If a branch leads to a leaf with a term t, the corresponding sequence of the inner node labels from the branch is exactly the *string representation* of the term t, i.e., the sequence of symbols obtained by traversing t in the left-to-right depth-first manner. Fig. 1 shows a perfect discrimination tree for the set $\{f(f(a)), g(f(x),x), g(f(a),a), g(f(a),b), g(h(a),a)\}$.

Maintenance of a perfect discrimination tree is straightforward regardless of the chosen concrete representation. If we need to insert a term t, we traverse it in the left-to-right depth-first manner. In parallel with the traversal of t, we traverse the discrimination tree by following at each step an edge leading to a node with the symbol currently being visited in the term. If we reach a leaf, it means that the term has already been integrated into the tree. If at some step we cannot find a node corresponding to the current symbol, we add a whole branch for the remaining part of term traversal, ending with a leaf. If we need to remove a term, we find a corresponding branch (if it exists) and cut off the nodes that do not lead to other terms.

Retrieval of instances is done by simultaneously traversing the query term and traversing the discrimination tree with backtracking. If the symbol in the current position in the query term is a

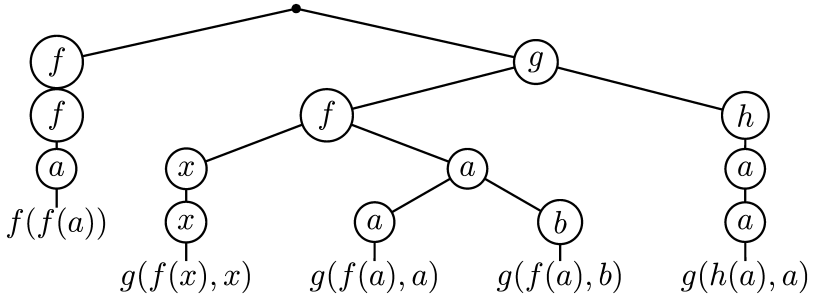

Fig. 1. Perfect discrimination tree for $\{f(f(a)), g(f(x),x), g(f(a),a), g(f(a),b), g(h(a),a)\}$.

function symbol, a forward move in the tree can only be done towards a node labeled by this function symbol. If we fail to find such a node, we have to backtrack to the most recent backtrack point. If the symbol in the current position in the query is an uninstantiated variable, the current position in the tree becomes a backtrack point, and we skip some part of the leftmost branch starting with this node. Labels from this part of the branch must represent a term which becomes the current instance of the variable. Upon backtracking, we follow the leftmost of the as yet unexplored branches. When a variable instantiated by a term s is met in the query, we treat it as if it was s itself. When a leaf is reached, we report its content as a found instance together with the collected substitution, and backtrack with the purpose of finding more instances. The described retrieval procedure provides perfect filtering for instances and delivers the relevant substitutions explicitly. For other variants of discrimination trees and related algorithms, see [8,3].

Since in experiments we are going to compare other indexing techniques for instance retrieval with our implementation of perfect discrimination trees, it is necessary to specify the concrete data representation we are using. Fig. 2 shows how the discrimination tree from the previous example is represented. Branches coming out of an inner node or the root are collected into a linked list. An element of the list contains the term symbol corresponding to the branch, and a pointer to the list corresponding to the subtrees of the branch. To detect the absence of need for backtracking as early as possible, the lists are ordered using some order on term symbols, in which any variable is smaller than any function symbol. Leaves contain a special symbol # instead of a term symbol, together with a pointer to the indexed term.

An alternative solution would be to represent branching with arrays (as in the unit equality prover WALDMEISTER, see, e.g. [10]) or hash tables. Such representation can speed up retrieval in trees with high branching factors but also consume a large amount of memory in the case of large signatures.

We have chosen the representation based on ordered linked lists mostly for its simplicity and cheap maintenance. Another reason is that all list nodes are small pieces of memory of the same size, and allocation of such pieces is usually easier as it is not hindered by memory fragmentation.

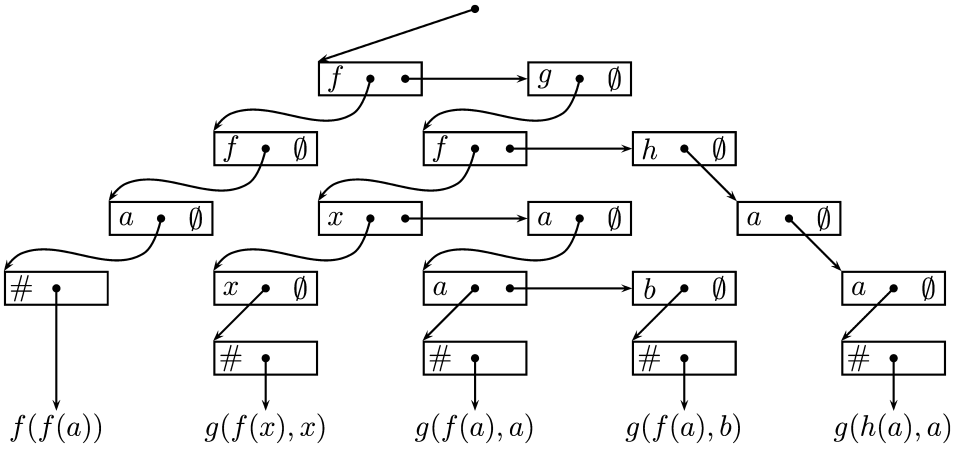

Fig. 2. Concrete representation of the discrimination tree from Fig. 1.

2.4. Standard path indexing

2.4.1. Paths in terms

The key notion for describing path indexing is that of a *path in a term*. Every path is a sequence of function symbols interleaved with non-negative integers and ending with a function symbol. Suppose that we represent a term as a tree whose nodes are labeled by the function symbols and variables, and the edges by the corresponding argument positions. For example, the term $f(g(a,x,b),h(y,x))$ is represented by the tree in Fig. 3. Now, if we collect the labels excluding variables and the numeric labels of the edges leading to nodes with variables, along any of the paths in the tree, we get a path in the original term. The paths in this term are the following sequences: $[f]$, $[f.1.g]$, $[f.1.g.1.a]$, $[f.1.g.3.b]$ and $[f.2.h]$. $Paths(t)$ will denote the set of all paths in a term t. We will sometimes speak about paths without associating them with a particular term, in which case we assume that the objects we call paths are paths in some terms of the background signature. A path in a term will be called *maximal*, if it is not a proper prefix of any other path in the term. The set of all maximal paths in t is denoted by $MaxPaths(t)$. For example, the term considered above contains three maximal paths: $[f.1.g.1.a]$, $[f.1.g.3.b]$ and $[f.2.h]$.

2.4.2. General scheme for path indexing

Standard path indexing is based on the following observation: every instance t of a term q *contains all paths* from q. This property allows us not to consider terms that do not have some paths from q, as potential instances of q. We can always restrict ourselves by considering only the maximal paths in q. Indeed, a term t contains all paths from q if and only if it contains all maximal paths from q.

We will now formulate a general scheme for path indexing. The index for a set of terms $\mathcal{I}$ should give us a possibility to extract the set $\mathcal{I}_\pi = \{t \in \mathcal{I} | \pi \in Paths(t)\}$ for every given path π. When we need to retrieve all instances of a term q from $\mathcal{I}$, we compute the *candidate set* containing all terms from $\mathcal{I}$ that contain all maximal paths from q:

$$CandSet(\mathcal{I}, q) = \bigcap_{\pi \in MaxPaths(q)} \mathcal{I}_\pi.$$

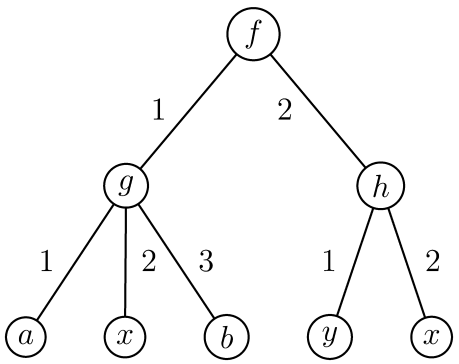

Fig. 3. Labeled tree for $f(g(a,x,b),h(y,x))$.

The candidate set $CandSet(\mathcal{I}, q)$ contains all instances of q in $\mathcal{I}$, and typically is much smaller than $\mathcal{I}$. Unfortunately, it may contain some terms that are not instances, since the presence of all paths from q in a term t does not guarantee that t is an instance of q. For example, $t = f(g(a,a),h(b,b))$ contains all paths from $q = f(g(a,x),h(y,x))$, but is not an instance of q, since t has different terms a and b in the positions corresponding to occurrences of the same variable x in q. Even if we know that $CandSet(\mathcal{I}, q)$ coincides with the set of instances, as in the case of a linear term q, it tells us nothing about the corresponding substitution, which we have to find explicitly. To resolve these problems, we need to perform additional *cleanup* on $CandSet(\mathcal{I}, q)$, getting rid of terms that are not instances of q and computing substitutions for those that are.

2.4.3. Implementation

Our implementation of the standard path indexing technique is a straightforward specialisation (for instance retrieval) of the implementation described in [8].

The sets $\mathcal{I}_\pi$ are represented as sorted lists of pointers to the datastructures representing the indexed terms, or any other objects that can serve as *term identifiers*. These lists will be called *path-lists*. We assume that all indexed terms and their subterms are *perfectly shared*, i.e., different pointers always represent syntactically different terms. Intersection of sorted lists can be computed according to the procedure *intersect* $(L_1, \ldots, L_n)$ in Fig. 4, which works roughly as follows. It descends to the first common element in L_1 and L_2, then tries to find it in L_3, L_4, and so on. If it is absent from one of the L_i, the element nearest to it is assigned as the current candidate and the search backtracks to L_1.

A minor difference with [8] is that we do not have to construct binary trees representing nested unions and intersections (aka *query trees* [3]) simply because we do not need to compute unions of path-lists at all.

For accessing the path-list, associated with a path, any hashing in which the path is the key will do. Another solution, which is probably optimal, is to employ the well-known datastructure called *trie*. In our settings, a trie is a directed tree whose edges are labeled by function symbols or non-negative integers. Every node with an incoming edge labeled by a function symbol corresponds to a path π in some terms from $\mathcal{I}$, and contains the pointer to the path-list $\mathcal{I}_\pi$. The correspondence is straightforward: π is the sequence of node labels in the path leading from the root of the trie to the node in question. Consider an example with a small indexed set $\mathcal{I} = \{t_1, t_2, t_3\}$, where $t_1 = f(a, g(a))$, $t_2 = f(a, g(b))$, and $t_3 = g(f(a,b))$. The corresponding trie is depicted in Fig. 5.

Given a single path π, to find $\mathcal{I}_\pi$, we can simply navigate through the trie, following the nodes, labeled with the corresponding symbols from π, until we reach the node containing $\mathcal{I}_\pi$. If we fail to find such a node in the trie, we assume that $\mathcal{I}_\pi$ is empty. Of course, we do not have to represent all paths in a query term explicitly. Instead, we traverse the trie simultaneously with traversing the term. Each time we visit a function symbol in the term or decide to examine its argument with a particular number, we follow the corresponding edge in the trie, possibly after memorising the current position for backtracking. Maintaining the index is also straightforward. To insert a term t, for every path π in it, we find the node with $\mathcal{I}_\pi$ and add t to this path-list. If there is no such node in the trie, we first add the corresponding branch to the trie with empty path-lists in the new nodes. To remove t, we delete it from all $\mathcal{I}_\pi$, where $\pi \in Paths(t)$. As soon as a path-list $\mathcal{I}_\pi$ becomes empty, it is safe to remove the corresponding node from the trie.

```
procedure intersect(L_1, ..., L_n)
  assume n > 1, L_1, ..., L_n are sorted in ascending order
  if L_1 = nil then stop
  cand := hd(L_1)
  i := 1
  loop
     while L_i ≠ nil and hd(L_i) < cand
        L_i := tl(L_i)
     if L_i = nil then stop
     if hd(L_i) = cand
      then
         L_i := tl(L_i)
         if i = n
          then
             report cand as a member of the intersection
             if L_1 = nil then stop
             cand := hd(L_1)
             i := 1
          else
             i := i + 1
      else
         cand := hd(L_i)
         i := 1
end
```

Fig. 4. Intersection of sorted lists.

Finally, we describe how the cleanup on the candidate set is implemented. As soon as the retrieval procedure for a query q reports a newly found candidate term t, we apply a matching procedure $MatchCand(q, t)$ to it, in order to check if t is an instance of q and to find the corresponding substitution. Although *MatchCand* can implement a general matching algorithm, for the sake of efficiency, in the algorithm shown in Fig. 6 we exploit the assumption that t has all paths from the term q.

2.5. Experiment settings

In the rest of the article we compare experimentally several implementations of instance retrieval. We use the COMPIT methodology (see [10]) to perform extensive and reproducible experiments with our implementations on realistic benchmarks. To make the benchmarks realistic, a real prover is run on real problems, and all operations related to instance retrieval are logged into a special file. To test a particular implementation, it is linked to a simple driver code. The driver reads the benchmark file and submits the index update and retrieval operations, together with all the necessary data, to the implementation in question. It also measures the time, taken by retrieval and index updates, memory, used by the index, collects various statistics on the operations, and prints it all as a Prolog fact, which makes it easy to maintain and process large databases of experimental results.

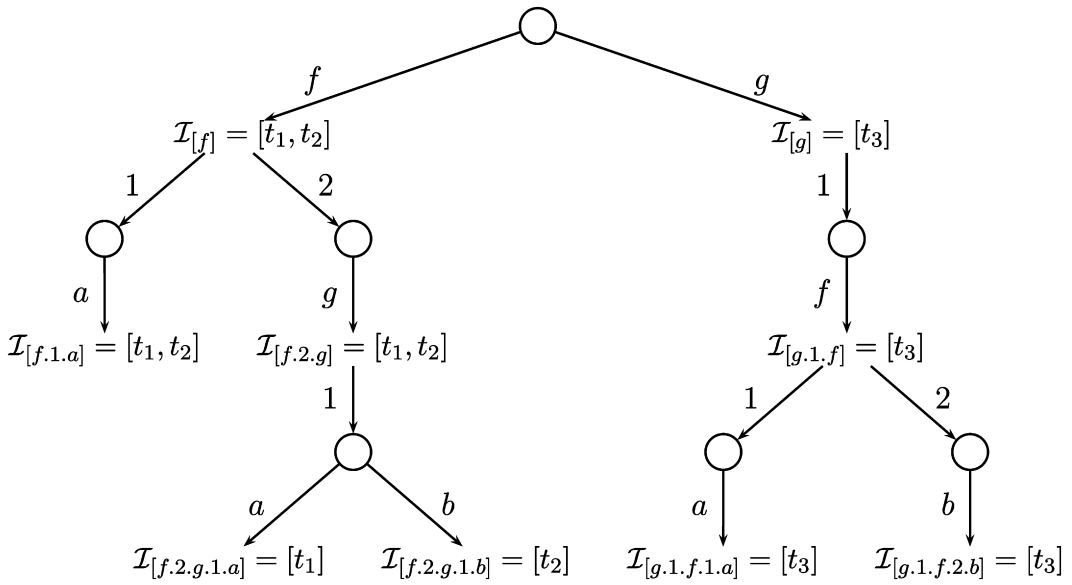

Fig. 5. Example of a trie for accessing path lists. $t_1 = f(a, g(a)), t_2 = f(a, g(b)), t_3 = g(f(a, b))$.

```
procedure MatchCand(q, t : term)
  θ : substitution := ∅
  if MatchCand'(q, t, θ)
    then report success with substitution θ
    else report failure
end

fun MatchCand'(q, t : term, var θ : substitution) : bool
  if q is a variable x
    then
      if x is instantiated in θ
        then return (xθ = t)
        else
          add assignment x → t to θ
          return true
    else
      let q = f(q_1, ..., q_n), t = f(t_1, ..., t_n)
      return ⋀_{i=1}^{n} MatchCand'(q_i, t_i, θ)
end
```

Fig. 6. Matching test for candidate cleanup.

Table 1
Benchmark suite profiles

	GEQ	P+UEQ
Total number of problems	2896	585
Average number of insertions	2,00,866	59,948
Average number of deletions	1,62,489	22,419
Average maximal index size	51,897	40,964
Average indexed term size	9	12
Average number of queries	15,018	33,577
Average query size	10	14
Average queried index size	26,578	22,858
Average number of instances per query	11	25

For our experiments, we use all the 2896 equality problems from the TPTP library [18], version 2.4.1. The benchmarks are generated by running our system VAMPIRE [13], v5.0, on a Pentium III machine, 1 GHz, 256 kb cache, running Linux, giving it 200 s time limit and 256 Mb memory limit. It is important that we are running VAMPIRE with the OTTER variant of the pick-given clause algorithm, which uses backward demodulation heavily. In the context of the DISCOUNT algorithm, used, for example, in VAMPIRE, WALDMEISTER [5] and E [15], the efficiency of instance retrieval is a minor issue due to fewer backward simplification steps.

Most of the time, we deal with the 585 pure equality (P+UEQ) problems, out of which 454 are unit equality problems. Additionally, in Section 8 we give a summary of results on the general class of problems with equality (GEQ). Table 1 presents the main quantitative characteristics of both benchmark suites. The characteristics are first calculated for every particular problem, and the corresponding figures in the table are obtained by dividing their sum by the number of participating problems.

The table exposes an important difference in requirements with search for generalisations used for forward demodulation, where there are usually several orders of magnitude more retrievals than maintenance operations. Almost all subterms in a new clause have to be inserted into the index for backward demodulation, and at most two of them can be used for retrieval. The situation with forward demodulation is completely opposite.

The main parameter of comparison is the overall time taken by both maintenance and retrieval. When two techniques are compared, we compare the sums of their running times on all problems from the benchmark suite in question, thus obtaining an average figure. We also compare memory usage, which is represented by two estimates: the lower and upper bound. The lower bound is the maximal amount of memory occupied by the index datastructures. This figure does not include memory that was used by the datastructures but later reclaimed. This corresponds to an ideal situation when all reclaimed memory can be reused by the index itself or other datastructures in the prover. Usually, this is not the case, and to balance the comparison we also calculate the upper bound. In this case we assume that the reclaimed memory can only be reused by the index itself, and the pieces of memory, reclaimed but not reused, are included in the estimate. Since in all our experiments the margin between the estimates turns out to be negligible (less than 1%), in what follows we only consider the upper bound.

All our implementations of indexing techniques are written in C++. The driver code is written in C, so it can be used for testing implementations both in C an C++.[1]

3. First experiments

We start our investigation by comparing the performance of our implementation of discrimination trees (Section 2.3) with the standard implementation of path indexing (Section 2.4.3). On our main benchmark suite, the P+UEQ problems, the path indexing implementation is inferior to discrimination trees, as it is 3.23 times slower, while uses 5% more memory (see Section 8, Tables 3 and 4). Nevertheless, the idea behind path indexing seems promising, and in the following three sections we describe optimisations that altogether make the standard path indexing technique much more attractive.

4. Better path-list representation

Our analysis of several randomly chosen execution profiles for the standard path indexing implementation indicated that, when the index is sufficiently large, *most of the time is spent on traversing the path-lists*. This traversal is needed for inserting terms into, and deleting them from the index, as well as for computing intersection of path-lists in retrieval attempts. Computing intersection amounts to checking elements of the lists one by one. In any retrieval attempt, we are bound to traverse at least one of the lists completely. In the worst case, all of them are traversed completely. Similarly, when inserting a term identifier into a path list, we have to skip one by one all smaller elements of the list. When the length of the path lists grows with the growth of the indexed set, such traversal becomes expensive.

To identify possible directions for optimisation, we consider the following example. For simplicity we assume that the term identifiers are integer numbers. Suppose that we want to intersect the list [11] with the list $[1, \ldots, 13]$, i.e., a list, containing all integers between 1 and 13. Before we come to the right position in the second list, we will have to check all its ten elements that are smaller than 11. This could probably be done faster if we were able to *jump over several elements* of the second list in one go. To this end we employ *skip lists* [11].

Fig. 7 shows an example of a skip list for representing the list of integers between 1 and 13. A skip list can be viewed as an ordered linked list whose nodes are equipped with additional pointers. In addition to the field for storing the key, a node in an ordered linked list contains a pointer to the next node. A node of a skip list may contain a whole array *next* of pointers to some nodes with greater values of keys. The size of the array is called the *height* of the node. The height of the node N_4 in Fig. 7 is 3. The first element of the array, *next* [0], always points at the next node in the list. If a pointer *next* $[i+1]$ is not null, then *next* $[i]$ is also not null, and if *next* $[i+1] \neq$ *next* $[i]$, then the node identified by *next* $[i]$

[1] The driver code, some of the implementations of indexing techniques discussed here, and the VAMPIRE source, modified for benchmark collection, are being reviewed by other COMPIT users, and the final versions will be available as parts of the COMPIT distribution package (http://www.mpi-sb.mpg.de/~hillen/compit). The originals are available from the authors.

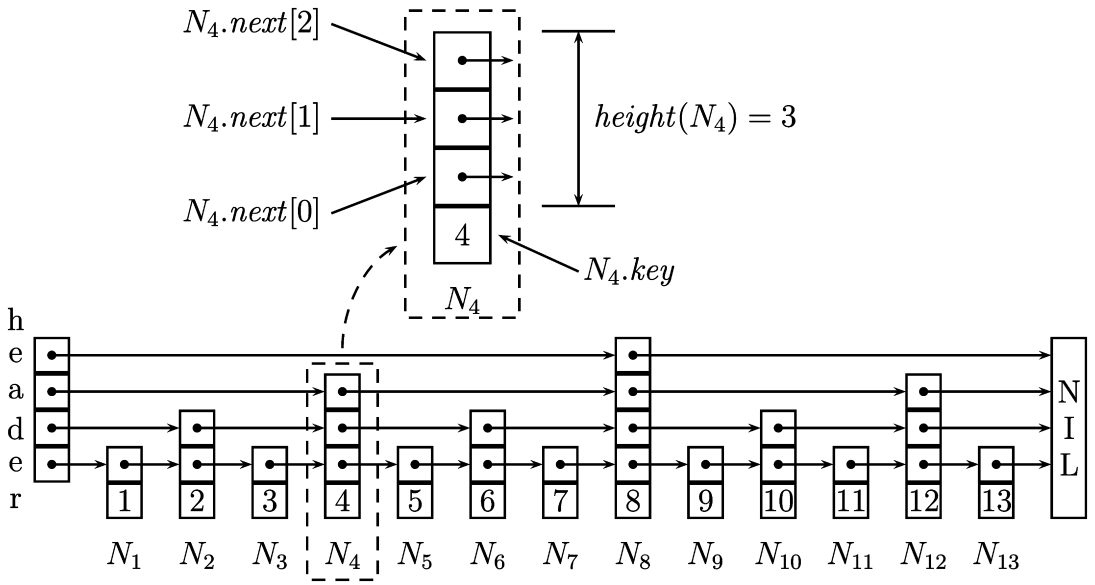

Fig. 7. Skip list for [1, 2, . . . , 12, 13].

precedes the one identified by *next* $[i+1]$ in the list. The height of a node, identified by a pointer *next* $[i+1]$, is always greater or equal to i. Moreover, it is the nearest node in the list with such height.

The pointers *next* $[i]$, where $i > 0$, are intended to facilitate far jumps when the list is traversed. The pointer $N_4.next$ [2] in Fig. 7 allows us to jump over three nodes: N_5, N_6, and N_7. Normally, the bigger i is, the farther from the current node we can jump by using the pointer *next* $[i]$.

In addition to nodes of various heights, every skip list has an array *header* of pointers to nodes. The size of the array should not be smaller than the maximal height of a node in the list. Every cell *header* $[i]$ contains the pointer to the first node in the list whose height is greater than i. Thus, *header* [0] always points to the very first node in the list.

A skip list is a probabilistic datastructure in the following sense. For optimal performance of search, the heights of the nodes are chosen randomly, but according to a certain distribution. Ideally, $\frac{1}{2}$ of all the nodes should be of the minimal height 1. Another $\frac{1}{4}$ of the nodes should be of the height 2. In general, the fraction of nodes with a height i should be $\frac{1}{2^i}$. Our implementation approximates this requirement by following the distribution when new nodes are added to a list. When a node is deleted, no adjustment is made to the remaining nodes. Since the height of a node is completely independent of the value of the key, we can not bias deletion towards some particular heights. Thus, deletion of nodes spoils the distribution very rarely.

Suppose now that we are looking for the first element in the skip list from Fig. 7, with a key greater than or equal to 10. First, we check the key in the node identified by the highest pointer in the header, i.e., *header*[3]. The key $8 < 10$, which means that we can immediately go to the node N_8, thus skipping seven nodes in one step. The highest pointer in the node N_8 is null, so we have to go one level down in $N_8.next$. Now $N_8.next[2] \to key = 12 > 10$, and we have to go one level down again. Finally, $N_8.next[1] \to key = 10$ and we can report success. Only minor adjustments

are needed to implement insertion and removal on the base of the search algorithm, as well as fast traversal needed for computing intersection of skip lists.

The expected cost of search in a skip list is logarithmic w.r.t. the length of the list (see [11]). This suggests that an accurate implementation of insertion, deletion and traversal on the base of the search algorithm may radically improve performance of the whole instance retrieval procedure. This hypothesis is supported by our experimental data. We have compared two implementations that differ only on the representation of path-lists on the P+UEQ problems. The implementation using skip lists turns out to be 12.6 times faster on average than the standard one using ordinary sorted lists, although the index requires 1.51 times more memory (for details, see Section 8, Tables 3 and 4). The memory consumption overhead can be explained by the need to store one additional pointer in the nodes of skip-lists on average, compared to the nodes of ordinary sorted lists.

5. Rearranging the order of path-list intersection

When running our implementations of path indexing on benchmarks with large indexed sets, we noticed that the path-lists participating in the intersection vary in size very much. Typically, the average size of the lists may be big, but at the same time there may be a few very short path-lists, corresponding to the paths that are rare in the indexed set. Long path-lists typically have relatively many elements in common, while short path-lists tend to have few. This observation gives rise to another optimisation.

Suppose we want to run our intersection procedure on the following five lists:

$L_1 = [1, 2, 3, \ldots, 100]$ (all integers from 1 to 100)
$L_2 = [1, 3, 5, \ldots, 99, 100]$ (odd integers from 1 to 99, and 100)
$L_3 = [2, 4, \ldots, 100]$ (even integers from 2 to 100)
$L_4 = [1, 100]$
$L_5 = [2, 100]$.

It turns out that the behaviour of our intersection procedure in Section 2.4.3 is very sensitive to the order in which the lists are arranged. First, let us see how *intersect* $(L_1, L_2, L_3, L_4, L_5)$ works. In the beginning of the outer loop, it finds a common element in L_1 and L_2, and makes it the current candidate. In our example, all the 51 elements of L_2 are eligible, and most of them are discarded at the next step as absent from the list L_3. By the end of the procedure execution, all elements of L_1, L_2 and L_3 will have been examined in some way. Representing the lists as skip lists does not help us much, since their main advantage, the possibility of far jumps, can not be used in these settings.

We can facilitate more far jumps if we *rearrange the order of lists, considering shorter lists before longer ones*. Execution of *intersect* $(L_4, L_5, L_3, L_2, L_1)$ quickly discovers that $L_4 \cap L_5$ consists of only one element 100, thus in the lists L_3, L_2 and L_1 we only need to make a few far jumps skipping all elements smaller that 100. Moreover, it often happens that the intersection of a few shortest path-lists is empty, and we can avoid processing the longer lists completely.

Experimental comparison on the P+UEQ problems shows that adding this heuristic to the implementation based on skip lists, makes it 1.6 times faster on average, and 20.4 times faster than the sorted list-based one (for details, see Section 8, Tables 3 and 4). Since the overhead for sorting the collections of path-lists by size is very small, the implementation with this heuristic never performs worse than the one without it.

6. Compiled cleanup

Now, when the retrieval of candidates has become relatively fast, a significant part of the overall time is often spent on filtering out the wrong candidates and extracting the substitutions, which makes it a potential target for optimisations. On cleanup, we call the procedure $MatchCand(q,t)$ (see Section 2.4.3) on many different candidate terms t for one fixed query term q. This suggests that we can use *run-time algorithm specialisation with compilation*.

The idea of *run-time specialisation of algorithms with compilation*, see, e.g. [19,14], is inspired by the use of *partial evaluation* in optimising program translation (see, e.g. [6]). Suppose that we need to execute some algorithm $alg(A,B)$ in a situation where a value of A is fixed for potentially many different values of B. To do this efficiently, we can try to find a specialisation of alg for every fixed A, i.e., such an algorithm alg_A, that executing $alg_A(B)$ is equivalent to executing $alg(A,B)$. The specialised algorithm may be more efficient than the generic one, since it can exploit some particular properties of the fixed value A. Typically, $alg_A(B)$ can avoid some operations that $alg(A,B)$ would have to perform, if they are known to be redundant for this particular parameter A. In particular, we can often identify some tests that are true or false for A, unroll loops and recursion, etc. The key difference between run-time specialisation and partial evaluation is that the values of A on which alg is specialised are not known statically, so the *specialisation takes place in run-time*. There is also an important technical difference. Partial evaluation is applied to algorithms explicitly represented as codes in some programming language. In run-time, we do not need any concrete representation of alg. We only have to *imagine alg* when we program the specialisation procedure. All we need is a concrete representation of the specialised version alg_A. This also means that we cannot use any universal methods for specialising algorithms, which is usually the case with partial evaluation. Instead, we have to program a specialisation procedure for every particular algorithm alg. An important advantage of doing so is that we can use some powerful ad hoc tricks exploiting peculiarities of alg and the representation of A and B, which are beyond the reach of any universal specialisation methods.

The specialised algorithm has to be represented in a form that can be interpreted, usually as a code of some abstract machine. Then the code itself can be additionally optimised by answer-preserving transformations that rely only on the semantics of instructions of the abstract machine. Instructions of the abstract machine can usually be represented as records. One field of such a record stores an integer tag that identifies the instruction type, other fields may be used for storing additional parameters of the instruction, for example a pointer to another instruction representing a label, if the semantics of the instruction requires a jump. All instructions of a code can be stored in an array, or list, or tree. Interpretation is done by fetching instructions in some order, identifying their type and executing the actions associated with this type. In C or C++ we can use a **switch** statement to associate some actions with different instruction tags. Modern compilers usually compile a **switch** statement with integer labels from a narrow range rather efficiently by storing the address of the statement corresponding to a value i in the ith cell of a special array. We exploit this by taking values for instruction tags from a small interval of integers.

Let us show how this method works for candidate cleanup. Suppose we are trying to match a query term $q = f(g(g(a,a),a),h(x_0),g(x_1,x_0))$ against some candidate term t which is known to contain all paths from q, and, therefore, is of the form $f(g(g(a,a),a),h(t^{(1)}),g(t^{(2)},t^{(3)}))$. The straightforward algorithm $MatchCand(q,t)$ uses no assumptions about q and t except the fact that all non-variable

positions in q agree with the corresponding positions in t. The execution of $MatchCand(q, t)$ requires traversing the terms q and t in parallel, and q has to be traversed completely. In particular, we have to traverse the ground subterm $g(g(a, a), a)$ both in q and t, which is redundant because we already know that t contains all paths of q. During traversal, we examine the top symbol of every visited subterm to see if it is a variable. This is also redundant, since we know the structure of q. Finally, for every occurrence of a variable in q we have to check if it has been instantiated. When we get to the first occurrence of x_0 in q, this test is redundant since we know that so far x_0 has not been considered at all. When we get to the second occurrence, the test is redundant again, since we know that x_0 has been instantiated when its first occurrence was visited.

These redundant actions can be avoided in the specialised version of $MatchCand$. The essence of the algorithm for $MatchCand_q(t)$ can be described in the following way. The procedure has to get to the subterms $t^{(1)}, t^{(2)}$ and $t^{(3)}$ as quickly as possible. As soon as we get both $t^{(1)}$ and $t^{(3)}$, we check that they are equal. When either $t^{(1)}$ or $t^{(3)}$ is known, we make it the instance of x_0 in the substitution θ, and as soon as $t^{(2)}$ is known, we make it the instance of x_1.

Let us see how $MatchCand_q(t)$ can be compiled. We rely on the following representation of indexed terms. A term $f(t_1, \ldots, t_n)$ is represented by a structure, containing f as the first field, together with an array arg of pointers identifying the subterms t_i. We assume that the terms are perfectly shared, that is, only one copy of every term is stored, then comparison of terms can be implemented by comparing the pointers. We also assume that query variables are identified by small integers shown as subscripts of x in our examples. The memory of our abstract machine will consist of a variable A for storing a pointer to an array of term arguments, and a sufficiently large array B for storing such pointers. The resulting substitution θ is stored as an array $subst$, such that $subst[i]$ is the pointer representing $x_i\theta$. Parameters of instructions denoted by p and r will be pointers to cells in B and $subst$ correspondingly, and those denoted by i are integer numbers. The list of basic instructions, together with their semantics, is given in Table 2.

Now, the algorithm for $MatchCand_{f(g(g(a,a),a),h(x_0),g(x_1,x_0))}(t)$ can be compiled into the sequence of instructions, shown in Fig. 8. In its turn, the code can be optimised in several ways. For example, instructions 6 and 7 can be swapped. This transformation makes sense because instruction 7 may fail, and in this case we do not have to execute instruction 6. Also, we can pack some subsequent instructions into bigger ones. For example, we can pack *Store* p and *Down* i into an instruction *StoreAndDown* p, i with the following semantics: $*p := A; A := A[i] \to arg$. This enables our implementation language compiler to use a machine register for storing the value of A used in the assignment $*p := A$ and for accessing $A[i]$ later. Additionally, this transformation reduces the total

Table 2
Instructions for the candidate cleanup abstract machine

Tag	Parameters	Semantics
Init		$A := t \to arg$
Store	p	$*p := A$
Restore	p	$A := *p$
Down	i	$A := A[i] \to arg$
Instantiate	r, i	$*r := A[i]$
Compare	r, i	**if** $*r \neq A[i]$**return failure**
Success		**return success**

			/* assume $t = f(g(g(a,a),a), h(t^{(1)}), g(t^{(2)}, t^{(3)}))$ */
0	*Init*		/* $A := (g(g(a,a),a), h(t^{(1)}), g(t^{(2)}, t^{(3)}))$ */
1	*Store*	$B+0$	/* $B[0] := (g(g(a,a),a), h(t^{(1)}), g(t^{(2)}, t^{(3)}))$ */
2	*Down*	1	/* $A := (t^{(1)})$ */
3	*Instantiate*	$subst+0,0$	/* $subst[0] := t^{(1)}$ */
4	*Restore*	$B+0$	/* $A := (g(g(a,a),a), h(t^{(1)}), g(t^{(2)}, t^{(3)}))$ */
5	*Down*	2	/* $A := (t^{(2)}, t^{(3)})$ */
6	*Instantiate*	$subst+1,0$	/* $subst[1] := t^{(2)}$ */
7	*Compare*	$subst+0,1$	/* **if** $t^{(1)} \neq t^{(3)}$ **return failure** */
8	*Success*		

Fig. 8. Code for $MatchCand_{f(g(g(a,a),a),h(x_0),g(x_1,x_0))}(t)$.

number of instructions, and thus the time spent on moving between instructions and examining their tags.

To assess the proposed technique experimentally, we have implemented it on top of the candidate retrieval implementation described in the previous section. The new implementation with compiled cleanup proved to be on average 1.2 times faster then the older one with straightforward cleanup, and 24.5 times faster than the initial implementation, on the P+UEQ problems (for details, see Section 8, Tables 3 and 4). The compilation-related overhead never leads to any noticeable slowdown.

7. Relational path indexing

So far we have only discussed some ways of efficiently implementing the same standard path indexing scheme, where retrieval of candidates containing all paths from the query is completely separated from the cleanup operations. Now we are going to describe a novel path indexing based technique, where the cleanup is tightly integrated with the candidate retrieval process. Roughly, the integration is done by interleaving path-list intersection operations with equality tests on subterms of the indexed terms. We would like to make it clear from the beginning that the main purpose of introducing the new technique is not only performance gains over the efficient implementation of standard path indexing, presented earlier. The proposed technique employs a very declarative description of retrieval with perfect filtering, and as such, it provides a logically clear and flexible base for more complex operations, like instance retrieval modulo commutativity and backward subsumption on multi-literal clauses, which will be discussed later. A detailed formal description of the proposed technique would be unnecessarily long, so we restrict ourselves to considering an example and exposing the logic behind it.

7.1. General scheme for retrieval

Every path π in a term t identifies a non-variable subterm t/π in t, whose top symbol coincides with the last symbol in the path. For example, $t/[f.1.g] = g(a,a)$ and $t/[f.3.h] = h(a,b)$, if

$t = f(g(a,a),a,h(a,b))$. Denote by arg_i, where i is a positive integer, the function on terms which returns the ith argument of a term. For example, $arg_3(f(x_0,a,h(x_1))) = h(x_1)$. Suppose that we have a query $q = f(g(x_0,x_0),a,h(x_1,x_0))$. Instances of q and corresponding substitutions can be characterised by the following condition: $t = q\theta$ if and only if $[f.1.g],[f.2.a],[f.3.h] \in Paths(t)$ and $x_0\theta = arg_1(t/[f.1.g]) = arg_2/(t/[f.1.g]) = arg_2(t/[f.3.h])$, and $x_1\theta = arg_1(t/[f.3.h])$. Thus, we can organise retrieval of instances of q from a term set $\mathcal{I}$ by simply selecting all terms in $\mathcal{I}$ satisfying this condition.

To give a finer description of the proposed technique, we need a notion of *path-relation* defined below, which replaces the notion of path-list. From now on, given a set of indexed terms $\mathcal{I}$ and a path π, $\mathcal{I}_\pi$ will denote a *path-relation* rather than a *path-list* for π. The word "relation" is used in the same sense as in relational databases, i.e., it means a finite collection of records with named attributes. If a path π ends in a function symbol f of arity n, the path-relation $\mathcal{I}_\pi$ will have an attribute **term** ranging over terms (term identifiers), and attributes $\mathbf{arg}_i$, whose values are also terms. For any term $t \in \mathcal{I}$, if the path π is in t and $t/\pi = f(t_1,\ldots,t_n)$, then the path-relation $\mathcal{I}_\pi$ will contain such a record r, that $r.\mathbf{term} = t$ and $r.\mathbf{arg}_1 = t_1,\ldots,r.\mathbf{arg}_n = t_n$.

As an example, consider a small indexed set $\mathcal{I} = \{t_1,t_2,t_3\}$, where $t_1 = f(g(a,a),a,h(b,a))$, $t_2 = f(g(a,b),a,h(b,b))$ and $t_3 = f(g(a,a),a,a)$. The path-relations for the paths $[f.1.g]$, $[f.2.a]$ and $[f.3.h]$ are shown in Fig. 9.

The analogy with relational databases can be extended to retrieval in the following way. Since $\mathcal{I}$ can be viewed as a relational database, we can describe the set of all substitutions corresponding to instances of a query q, as an SQL query. For example, the query $q = f(g(x_0,x_0),a,h(x_1,x_0))$ translates into the SQL query depicted in Fig. 10. To evaluate this query, we build the Cartesian product of the path-relations, specified in the **from**-clause, throw away those records that do not satisfy the **where**-clause and, finally, build the answer table with two attributes corresponding to the columns $\mathcal{I}_{[f.1.g]}.\mathbf{term}$, $\mathcal{I}_{[f.1.g]}.\mathbf{arg}_1$ and $\mathcal{I}_{[f.3.h]}.\mathbf{arg}_1$ in the Cartesian product. The value of the **instance** attribute in the answer table contains a retrieved instance, and the values for $\mathbf{x_0}$ and $\mathbf{x_1}$ represent the corresponding substitution. Of course, this is only a declarative view. One would almost certainly use a more efficient process in practice. The **where**-clause in the query contains a number of equality conditions on the attributes of the input tables. The conditions on the **term** attribute correspond to candidate retrieval, since they ensure that extracted indexed terms have all the paths from the query. The conditions on $\mathbf{arg}_i$ implement the necessary cleanup, as they filter out the candidates that have different terms under the paths leading to equal variables in the query. In the relational database terminology, such constraints are called *join conditions*, and the answer table is called a

$\mathcal{I}_{[f.1.g]}$:

term	$\mathbf{arg}_1$	$\mathbf{arg}_2$
t_1	a	a
t_2	a	b
t_3	a	a

$\mathcal{I}_{[f.2.a]}$:

term
t_1
t_2
t_3

$\mathcal{I}_{[f.3.h]}$:

term	$\mathbf{arg}_1$	$\mathbf{arg}_2$
t_1	b	a
t_2	b	b

Fig. 9. Path relations for $\mathcal{I} = \{t_1,t_2,t_3\}$, where $t_1 = f(g(a,a),a,h(b,a))$, $t_2 = f(g(a,b),a,h(b,b))$ and $t_3 = f(g(a,a),a,a)$.

$$
\begin{array}{l}
\underline{\textbf{select}}\ \mathcal{I}_{[f.1.g]}.\textbf{term instance},\ \ \mathcal{I}_{[f.1.g]}.\textbf{arg}_1\ \mathbf{x_0},\ \ \mathcal{I}_{[f.3.h]}.\textbf{arg}_1\ \mathbf{x_1} \\
\underline{\textbf{from}}\ \mathcal{I}_{[f.1.g]},\ \mathcal{I}_{[f.2.a]},\ \mathcal{I}_{[f.3.h]} \\
\quad \underline{\textbf{where}} \\
\quad\quad \mathcal{I}_{[f.1.g]}.\textbf{term} = \mathcal{I}_{[f.2.a]}.\textbf{term} = \mathcal{I}_{[f.3.h]}.\textbf{term} \\
\quad\quad\quad \underline{\textbf{and}} \\
\quad\quad \mathcal{I}_{[f.1.g]}.\textbf{arg}_1 = \mathcal{I}_{[f.1.g]}.\textbf{arg}_2 \quad \underline{\textbf{and}} \quad \mathcal{I}_{[f.1.g]}.\textbf{arg}_1 = \mathcal{I}_{[f.3.h]}.\textbf{arg}_2
\end{array}
$$

Fig. 10. Relational query for $f(g(x_0, x_0), a, h(x_1, x_0))$.

join of the input relations. The proposed technique, which we call *relational path indexing*, provides perfect filtering for instance retrieval, provided that all the necessary path-relations participate in the join and all the necessary conditions are present in the **where**-clause.

7.2. Implementations

In all our implementations of relational path indexing, the indexing datastructure is very similar to that for the standard variant of path indexing, described earlier. We are using the same trie to access the path-relations. The path-relations themselves are represented by skip lists, whose nodes contain an indexed term as the key for ordering. Unlike in the implementation of standard path indexing, the nodes of skip lists also store an additional pointer to the array of immediate subterms of the subterm, identified by the path. Using the indexed term as a key allows efficient joins on the **term** attribute. They are computed in the same way as intersection of path-lists represented by skip lists. When a record of a path-relation has been joined to others by the **term** attribute, the additional pointer can be used to access the values of $\textbf{arg}_i$.

7.2.1. Simple implementation of retrieval

As a starting point, we have implemented retrieval in a way similar to the one used in the standard technique. The join on **term** and the cleanup are separated. Roughly speaking, we first compute (in a lazy manner) the join on **term**, and for each record from the join we check the cleanup conditions.

7.2.2. Early cleanup

The relational description of a query is declarative, it does not impose any constraints on how the joins are computed. In particular, the order in which they are computed does not affect the result. We can exploit this by doing some cleanup operations before some operations needed to build the join on **term**. Let us consider the query $q = f(g(x_0, x_0), a, h(x_1, x_0))$ from the example above. As soon as we know that some term t has the path $[f.1.g]$, we can check the condition $arg_1(t/[f.1.g]) = arg_2(t/[f.1.g])$. If this inexpensive test fails, we avoid execution of operations needed to check that $[f.2.a], [f.3.h] \in Paths(t)$. Similarly, when both $[f.1.g]$ and $[f.3.h]$ are known to be in t, we can immediately check $arg_1(t/[f.1.g]) = arg_2(t/[f.3.h])$. In general, we check a cleanup condition as soon as the participating records become available as a result of joining on the **term** attribute.

7.2.3. Combining joins with compiled cleanup

The example considered above hides one inherent drawback of relational path indexing. In the query $q = f(g(x_0, x_0), a, h(x_1, x_0))$ all variable occurrences are arguments of subterms, identified by

maximal paths. This is not the case in general, so we can no longer restrict ourselves to considering only maximal paths. Any path in a query leading to a subterm containing some variables as immediate subterms, contributes to the join. For example, in $q = g(x_0, g(x_0, g(x_1, a)))$, instead of simply taking all terms from $\mathcal{I}_{[g.2.g.2.g.2.a]}$ as candidates, we have to access corresponding records in $\mathcal{I}_{[g]}, \mathcal{I}_{[g.2.g]}$ and $\mathcal{I}_{[g.2.g.2.g]}$, in order to find possible instantiatons of the query variables. Although dealing with such paths amounts to relatively efficient search in skip lists, it is too often beaten on performance by compiled cleanup. It is natural to try combining the advantages of compiled cleanup and early cleanup. We have implemented this idea in the following opportunistic way. Only those occurrences of variables in the query that are under maximal paths, give rise to join conditions that can be checked early. No additional path-relations need to be considered to check these conditions. If there are other variables, the join built in such a way may contain bad candidates filtered out by the additional compiled cleanup in the same way as it is done for standard path indexing.

7.3. Experiments

We have compared experimentally the three implementations, presented here against our best implementation of standard path indexing, which uses compiled cleanup, on the P+UEQ problems. All the new implementations use the same amount of memory, which is on average 27% more than the amount used by the fast standard path indexing. The first implementation (without early cleanup) is also 1.35 times slower. This is still an encouraging result. We can afford a moderate drop in performance, since we are mostly aiming at finding a flexible base for future extensions. The second implementation, employing early cleanup, is only 1.14 times slower. The third implementation, combining early and compiled cleanup, is already 1.11 times faster than the standard technique with compiled cleanup, and this figure shows the pure value of the early cleanup optimisation. This implementation is 27.2 times faster than the unoptimised implementation of standard path indexing, and is currently the fastest one we have. More experimental data is given in Section 8, Tables 3 and 4.

7.4. Instance retrieval modulo commutativity

Now we are going to show how relational path indexing can be used for instance retrieval modulo commutativity of some binary functions. For simplicity, we assume that the only commutative function symbols in the signature is $+$. *Syntactic equality* $=_C$ on terms *modulo commutativity* is defined in the following way:

$$s =_C t \Leftrightarrow \begin{cases} & s = t, \\ \textbf{or} & s = f(s_1, \ldots, s_n), t = f(t_1, \ldots, t_n), f \neq +, s_i =_C t_i \qquad \text{for } i = 1, \ldots, n, \\ \textbf{or} & s = +(s_1, s_2), t = +(t_1, t_2), s_1 =_C t_1, s_2 =_C t_2, \\ \textbf{or} & s = +(s_1, s_2), t = +(t_1, t_2), s_1 =_C t_2, s_2 =_C t_1. \end{cases}$$

Any term, equal modulo commutativity to a term t, will be called a *rotation* of t.

Term t is an *instance* of s *modulo commutativity*, if $t =_C s\theta$ for some substitution θ. Substitutions θ_1 and θ_2 are *equivalent modulo commutativity*, if $x\theta_1 =_C x\theta_2$ for any variable x. Now *instance retrieval modulo commutativity* is the following algorithmic problem. Suppose that we are given a term set $\mathcal{I}$, whose elements are pairwise distinct modulo commutativity. For any query term q, find

all such substitutions θ, that $q\theta =_C t$ for some term $t \in \mathcal{I}$. Obviously, we can restrict ourselves by considering only substitutions that are pairwise inequivalent modulo commutativity.

One can try to solve this problem by adapting an indexing technique for ordinary instance retrieval. For example, a possible solution is to store all rotations of terms from $\mathcal{I}$ in the index. In most cases this is not acceptable, as the index grows very quickly: every term with n occurrences of commutative symbols may potentially have 2^n different rotations. Alternatively, we can keep the index as it is, and try all rotations of the query for retrieval, provided the retrieval algorithm is adjusted to accommodate syntactic equality modulo commutativity. In this case retrieval becomes costly, as the number of its invocations grows drastically. Fortunately, the idea of relational path indexing allows us to formulate an elegant solution which is free from such drawbacks and is likely to be efficient in practice.

The basic idea behind the proposed technique is as follows. The arguments of a commutative functor can permute freely, so we do not distinguish the argument numbers, following commutative symbols in the paths. This observation is made in the conclusion of [8], and is also used in [3] for defining a filter for instance retrieval modulo associativity and commutativity. As long as only commutativity is concerned, we go slightly further by providing a perfect filter.

A path π is called *normalised*, if every argument number immediately following a commutative symbol in π is equal to 1. With every path π we associate a normalised path *norm* (π), which is obtained from π by replacing the corresponding argument numbers by 1. Additionally, we consider *denormalising* bit vector *denorm* (π), such that *denorm* $(\pi)[i] = 0$ if and only if the ith occurrence of a commutative symbol in π is followed by 1. For example, *norm* $([+.2.+.1.+.2.a]) = [+.1.+.1.+.1.a]$ and *denorm* $([+.2.+.1.+.2.a]) = 101$. It is obvious that any path π can be restored from *norm* (π) and *denorm* (π).

The indexing datastructure for a term set $\mathcal{I}$ consists of path-relations $\mathcal{I}_\pi$ for every normalised path π. Every path-relation $\mathcal{I}_\pi$ has the attribute **term** ranging over terms, and an attribute **denorm**, whose values are bit vectors. If a path π ends in a commutative symbol, the path-relation $\mathcal{I}_\pi$ has a term-valued attribute **arg**, otherwise it has attributes $\mathbf{arg}_i$, as before. Every path π in a term $t \in \mathcal{I}$, ending in a commutative symbol, generates two records r_1, r_2 in the path-relation $\mathcal{I}_{norm\ (\pi)}$ with the following values: $r_1.\mathbf{term} = r_2.\mathbf{term} = t, r_1.\mathbf{arg} = arg_1(t/\pi), r_2.\mathbf{arg} = arg_2(t/\pi), r_1.\mathbf{denorm} = denorm\ (\pi) \cdot 0$ and $r_2.\mathbf{denorm} = denorm\ (\pi) \cdot 1$, where $\cdot$ denotes concatenation. If π ends in a non-commutative symbol, it generates one record r in $\mathcal{I}_{norm\ (\pi)}$, where $r.\mathbf{term} = t, r.\mathbf{arg}_i = arg_i(t/\pi)$ and $r.\mathbf{denorm} = denorm\ (\pi)$.

For example, consider the term $t = +(+(a, f(b, c)), +(c, f(a, b)))$. The tree representation of this term is shown in Fig. 11. The entries in $\mathcal{I}_{[+.1.+]}$ and $\mathcal{I}_{[+.1.+.1.f]}$ generated by this term are shown in Fig. 12.

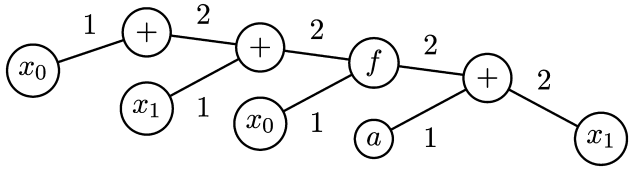

Fig. 11. Tree representation of $t = +(+(a, f(b, c)), +(c, f(a, b)))$.

$\mathcal{I}_{[+.1.+]}$:

term	**arg**	**denorm**
⋮	⋮	⋮
t	a	00
t	$f(b,c)$	01
t	c	10
t	$f(a,b)$	11
⋮	⋮	⋮

$\mathcal{I}_{[+.1.+.1.f]}$:

term	$\mathbf{arg}_1$	$\mathbf{arg}_2$	**denorm**
⋮	⋮	⋮	⋮
t	b	c	01
t	a	b	11
⋮	⋮	⋮	⋮

Fig. 12. Some path-relation records for $t = +(+(a, f(b,c)), +(c, f(a,b)))$.

$$
\begin{array}{l}
\underline{\textbf{select}}\ \mathcal{R}_1.\mathbf{term}\ \mathbf{instance},\ \mathcal{R}_1.\mathbf{arg}\ \mathbf{x_0},\ \mathcal{R}_2.\mathbf{arg}\ \mathbf{x_1} \\
\underline{\textbf{from}}\ \mathcal{I}_{[+]}\ \mathcal{R}_1,\ \ \mathcal{I}_{[+.1.+]}\ \mathcal{R}_2,\ \ \mathcal{I}_{[+.1.+.1.f]}\ \mathcal{R}_3,\ \ \mathcal{I}_{[+.1.+.1.f.2.+.1.a]}\ \mathcal{R}_4,\ \ \mathcal{I}_{[+.1.+.1.f.2.+]}\ \mathcal{R}_5 \\
\quad \underline{\textbf{where}} \\
\qquad \mathcal{R}_1.\mathbf{term} = \mathcal{R}_2.\mathbf{term} = \mathcal{R}_3.\mathbf{term} = \mathcal{R}_4.\mathbf{term} = \mathcal{R}_5.\mathbf{term} \\
\qquad\quad \underline{\textbf{and}} \\
\qquad \mathcal{R}_1.\mathbf{denorm}|_1 \neq \mathcal{R}_2.\mathbf{denorm}|_1 \\
\qquad\quad \underline{\textbf{and}} \\
\qquad \mathcal{R}_2.\mathbf{denorm}|_1 = \mathcal{R}_3.\mathbf{denorm}|_1 \\
\qquad\quad \underline{\textbf{and}} \\
\qquad \mathcal{R}_2.\mathbf{denorm}|_2 \neq \mathcal{R}_3.\mathbf{denorm}|_2 \\
\qquad\quad \underline{\textbf{and}} \\
\qquad \mathcal{R}_3.\mathbf{denorm}|_2 = \mathcal{R}_4.\mathbf{denorm}|_2 = \mathcal{R}_5.\mathbf{denorm}|_2 \\
\qquad\quad \underline{\textbf{and}} \\
\qquad \mathcal{R}_4.\mathbf{denorm}|_3 \neq \mathcal{R}_5.\mathbf{denorm}|_3 \\
\qquad\quad \underline{\textbf{and}} \\
\qquad \mathcal{R}_1.\mathbf{arg} =_C \mathcal{R}_3.\mathbf{arg}_1 \quad \underline{\textbf{and}} \quad \mathcal{R}_2.\mathbf{arg} =_C \mathcal{R}_5.\mathbf{arg}
\end{array}
$$

Fig. 13. Relational query for $+(x_0, +(x_1, f(x_0, +(a, x_1))))$.

The query term $+(x_0, +(x_1, f(x_0, +(a, x_1))))$ can be translated into a relational query as shown in Fig. 13. The logic behind this translation is as follows. Any instance t of the query is a rotation of some term of the form $+(t^{(1)}, +(t^{(2)}, f(t^{(3)}, +(a, t^{(4)}))))$, where $t^{(1)} =_C t^{(3)}$ and $t^{(2)} =_C t^{(4)}$. As such, t must have some paths $\pi_1 = [+]$, $\pi_2 = [+.i_1.+]$, $\pi_3 = [+.i_2. + .i_3.f]$, $\pi_4 = [+.i_4. + .i_5.f.2. + .i_6.a]$ and $\pi_5 = [+.i_7. + .i_8.f.2.+]$, where all $i_k \in \{1, 2\}$. This restriction is represented by the join condition on the **term** attribute in the relational query. The condition $\mathcal{R}_1.\mathbf{denorm}|_1 \neq \mathcal{R}_2.\mathbf{denorm}|_1$ reflects the requirement that $t^{(1)}$ and $+(t^{(2)}, f(t^{(3)}, +(a, t^{(4)})))$ correspond to *different arguments* of the top

function symbol in t. The operation $|_n$ on bit vectors is defined as follows: for a bit vector b, $b|_n$ denotes its subvector consisting of the first n bits of b. Since the paths π_2 and π_3 lead inside the same immediate subterm of t, the numbers i_1 and i_2 must coincide, which explains the join condition $\mathcal{R}_2.\mathbf{denorm}|_1 = \mathcal{R}_3.\mathbf{denorm}|_1$. Other conditions on **denorm** are explained in the same manner. Finally, the conditions $t^{(1)} =_C t^{(3)}$ and $t^{(2)} =_C t^{(4)}$ give rise to the last line of the **where**-clause.

We believe that the proposed scheme can be efficiently implemented. All our optimisations, discussed so far, are transferable to some extent. We can represent path-relations as skip lists, to facilitate fast join on the **term** attribute. Clusters of records in a path-relation with the same values of the **term** attribute can be collected into lists. Checking the cleanup join conditions early is possible at least for the paths not containing commutative symbols. Moreover, it is likely that early checks of the conditions on the **denorm** vectors can also help. Comparing prefixes of the **denorm** vectors can be done cheaply by using bitwise shifts, if the vectors are properly represented. Compiled cleanup is directly applicable to the paths, not containing commutative symbols, and for others the code can use the values of **denorm** to direct traversal of a candidate term. Testing syntactic equality of terms modulo commutativity should not cause many problems. In the simplest case, the indexed terms and their subterms can be stored in a normalised form, unique for all terms, equal modulo commutativity. Alternatively, we can associate with every term a pointer to its normal form. In both cases, the equality check is done in small constant time.

7.5. Backward subsumption on multi-literal clauses

Now we are going to show how the relational path indexing scheme can be adapted to solving another hard algorithmic problem: backward subsumption. The *subsumption relation* is defined on arbitrary first-order clauses, which can be viewed as multisets of literals, in the following way. A clause C *subsumes* a clause D, if there exists a substitution θ, such that $C\theta$ is a submultiset of D. For example, the clause $p(x, y) \vee q(y, z)$ subsumes the clause $p(a, b) \vee q(b, c) \vee r$, and the required substitution is $\{x \to a, y \to b, z \to c\}$. We can reduce the search space in a saturation-based prover by applying the so-called *backward subsumption*. When a new clause is added to the current clause multiset, we can discard those old clauses that are subsumed by the new one. Backward subsumption is inherently difficult since even checking subsumption of one clause by another is NP-complete [7].

As a retrieval condition, backward subsumption amounts to finding all clauses in a clause multiset $\mathcal{I}$, subsumed by a given query clause Q. This problem is related to instance retrieval, since every literal in the query has an instance in a subsumed clause. So, it is natural to try to adapt an instance retrieval technique to the new problem. Here we describe a solution to the backward subsumption problem, based on relational path indexing, as implemented in our system VAMPIRE ([13]). Due to its importance and difficulty, backward subsumption is the subject of a separate forthcoming publication. The description below is superficial and only intended to illustrate the flexibility of our approach.

First of all, let us show how the index for a clause set $\mathcal{I}$ is constructed. The indexed clauses are considered as sequences, rather then multisets. Only one sequence representation of a clause is used for indexing, and the choice of it is irrelevant. The notion of a path changes slightly. We are going to speak only about paths in literals, so the first element in a path is now always a *literal header*, i.e., a predicate with polarity, rather than a function symbol. As in the case of instance retrieval, the index provides a mapping of all paths π into corresponding path-relations $\mathcal{I}_\pi$. The path-re-

lations have an attribute **cl** ranging over clause identifiers, attributes $\mathbf{arg}_i$ as in instance retrieval, and an integer-valued attribute **lit**. Let C be a clause $L_1 \vee \cdots \vee L_n$. If a literal L_j in C contains a path π, the path-relation $\mathcal{I}_\pi$ should have a record r with the following values: $r.\mathbf{cl} = C$, $r.\mathbf{lit} = j$ and $r.\mathbf{arg}_i = arg_i(L_j/\pi)$. For example, consider the clause $C = p(f(a,b)) \vee p(f(b,c)) \vee \neg p(f(c,d))$. The entries in $\mathcal{I}_{[p.1.f]}$ and $\mathcal{I}_{[\neg p.1.f]}$ generated by this clause are shown in Fig. 14.

With such an index we can perform retrieval of subsumed clauses by following our general scheme: by translating query clauses into relational queries. For example, consider a query clause $Q = p(f(x_0), g(x_1)) \vee p(f(x_1), g(x_2)) \vee \neg p(x_0, a)$, which translates into the SQL query in Fig. 15. The translation is justified by the following considerations. Every clause subsumed by Q must have three different literals $p(f(t^{(1)}), g(t^{(2)}))$, $p(f(t^{(3)}), g(t^{(4)}))$ and $p(t^{(5)}, a)$, where $t^{(1)} = t^{(5)}$ and $t^{(2)} = t^{(3)}$. As such, it must contain the paths $[p.1.f]$, $[p.2.g]$, $[\neg p]$ and $[\neg p.2.a]$, moreover the first and second paths must occur at least twice. This requirement translates into the join condition on the **cl** attribute. Two of the literals above have both $[p.1.f]$ and $[p.2.g]$, and the remaining one has both $[\neg p]$ and $[\neg p.2.a]$. This gives rise to the second line in the **where**-condition. All the tree literals are different elements of the subsumed clause (although, may be syntactically equal), and this is reflected by the third line in the **where**-condition. The last line realises the requirements $t^{(1)} = t^{(5)}$ and $t^{(2)} = t^{(3)}$.

$\mathcal{I}_{[p.1.f]}$:

cl	$\mathbf{arg}_1$	$\mathbf{arg}_2$	**lit**
⋮	⋮	⋮	⋮
C	a	b	1
C	b	c	2
⋮	⋮	⋮	⋮

$\mathcal{I}_{[\neg p.1.f]}$:

cl	$\mathbf{arg}_1$	$\mathbf{arg}_2$	**lit**
⋮	⋮	⋮	⋮
C	c	d	3
⋮	⋮	⋮	⋮

Fig. 14. Some path-relation records for $C = p(f(a,b)) \vee p(f(b,c)) \vee \neg p(f(c,d))$.

```
select R1.cl subsumed_clause
from I[p.1.f] R1, I[p.2.g] R2, I[p.1.f] R3, I[p.2.g] R4, I[¬p] R5, I[¬p.2.a] R6
  where
    R1.cl = R2.cl = R3.cl = R4.cl = R5.cl = R6.cl
      and
    R1.lit = R2.lit  and  R3.lit = R4.lit  and  R5.lit = R6.lit
      and
    R1.lit ≠ R3.lit  and  R1.lit ≠ R5.lit  and  R3.lit ≠ R5.lit
      and
    R1.arg1 = R5.arg1  and  R2.arg1 = R3.arg1
```

Fig. 15. Relational query for $p(f(x_0), g(x_1)) \vee p(f(x_1), g(x_2)) \vee \neg p(x_0, a)$.

8. More experiments

To give the reader the big picture, we start this section with a summary of results of the experiments discussed in different parts of the article. Table 3 shows the resource usage by different implementations, relative to our implementation of discrimination trees. To give the reader a feeling of complexity of the used benchmarks, we also mention the average time and memory usage per problem. The names of techniques are abbreviated in the following way. When the standard scheme is used, which separates candidate retrieval from cleanup, the name starts with **std**. If an implementation relies on skip lists (Section 4) rather than sorted ones, **sl** is added to the name, **srtp** means that sorting of path-lists (Section 5) is used , and **ccu** stands for "compiled cleanup" (Section 6). The implementations, based on relational queries (Section 7) contain **rel** in their names. Please, notice that every such implementation employs skip lists and sorting of path-relations. Early cleanup (Section 7.2.2) is indicated by **ecu**. The used implementation of discrimination trees is called **disc**.

So far we have only experimented with pure equality problems. It is natural to ask whether the results are transferable to arbitrary problems with equality. The answer is "not quite," as demonstrated by Table 4. The table indicates that the performance gains from all our optimisations, except the use of skip lists, are rather modest compared to the ones in the case of pure equality problems. There is a simple explanation to this effect. The introduction of skip lists is the only optimisation that affects the speed of updates, the others improve only retrieval, often at the expense of more complex updates. The P+UEQ problems are characterised by relatively high frequency of retrieval attempts. On average, only 2.5 update operations are performed per retrieval attempt, while for GEQ the corresponding value is 24.2 (see Table 1). On GEQ the share of index updates in the overall time for **std+sl** and **rel** is 34 and 43% correspondingly, while on P+UEQ it is 4 and 6%, leaving much more space for retrieval-oriented optimisations.

Finally, we would warn the reader against drawing from our results a conclusion that path indexing is always preferable to discrimination trees. We can easily imagine a situation when memory usage or index maintenance time is critical. Discrimination trees is a universal technique, which means that the same index structure can support more than one retrieval relation, thus saving on memory and updates. Substitution trees [4] or context trees [2] could be another technique worth

Table 3
Experimental results on P+UEQ

impl	$\frac{\textbf{time (disc)}}{\textbf{time (impl)}}$	$\frac{\textbf{memory (disc)}}{\textbf{memory (impl)}}$	Average **time (impl)**, s	Average **memory (impl)**, Mb
Discrimination trees	1.00	1.00	91.3	3.1
std	0.31	0.95	291.5	3.3
std+sl	3.95	0.63	23.1	4.9
std+sl+srtp	6.33	0.63	14.4	4.9
std+sl+srtp+ccu	7.58	0.63	12.1	4.9
rel	5.60	0.50	16.3	6.2
rel+ecu	6.64	0.50	13.8	6.2
rel+ecu+ccu	8.43	0.50	10.8	6.2

Table 4
Experimental results on GEQ

impl	$\frac{\textbf{time (disc)}}{\textbf{time (impl)}}$	$\frac{\textbf{memory (disc)}}{\textbf{memory (impl)}}$	Average **time (impl)**, s	Average **memory (impl)**, Mb
Discrimination trees	1.00	1.00	23.0	3.4
std	0.07	0.98	309.8	3.5
std+sl	2.94	0.63	7.8	5.3
std+sl+srtp	3.89	0.63	5.9	5.3
std+sl+srtp+ccu	4.33	0.63	5.3	5.3
rel	3.57	0.52	6.4	6.5
rel+ecu	3.91	0.52	5.9	6.5
rel+ecu+ccu	4.48	0.52	5.1	6.5

trying when memory usage is critical, although they are unlikely to be very fast due to high maintenance cost.

9. Related and future work

The path indexing method was introduced by Stickel [17] and Ramesh et al. [12]. In [8], W. McCune compares experimentally an implementation of path indexing (for several retrieval relations including instance retrieval) corresponding closely to what we call standard implementation, with a number of variants of discrimination trees. In the book of Graf [3], a wide range of path indexing techniques is thoroughly described for most interesting retrieval relations, together with a number of implementation tricks. In particular, a combination of candidate retrieval with cleanup in one procedure, *extended path indexing*, is discussed, which unfortunately features very heavy maintenance operations (all pairs of equal subterms in an inserted/removed term have to be identified), and excessive memory requirements. Also, a promising technique is formulated for dealing with retrieval modulo associativity and commutativity.

The work, described in this article, can be continued in the following directions. We are preparing a thorough experimental assessment for the backward subsumption implementation outlined in Section 7.5, and, hopefully, a publication will follow, describing the technique in more detail. The technique for instance retrieval modulo commutativity (Section 7.4) is to be implemented, and a COMPIT-style infrastructure for experiments is to be created. We also believe that efficient indexing techniques can be constructed for instance retrieval modulo associativity and commutativity along the lines of relational path indexing. Another promising direction is to try to improve the behaviour of the variation of the standard technique, which restricts the length of a path (see [8]) and, thus, uses less memory. We believe that some improvement may be obtained by using compiled cleanup as in Section 6.

References

[1] N. Dershowitz, D. Plaisted, Rewriting, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, vol. 1, Elsevier Science, Amsterdam, 2001, pp. 533–608 (Chapter 9).

[2] H. Ganzinger, R. Nieuwenhuis, P. Nivela, Context trees, in: R. Goré, A. Leitsch, T. Nipkow (Eds.), First International Joint Conference on Automated Reasoning, IJCAR 2001, Lecture Notes in Artificial Intelligence, vol. 2083, Springer, Berlin, 2001, pp. 242–256.
[3] P. Graf, Term indexing, in: Lecture Notes in Computer Science, vol. 1053, Springer, Berlin, 1996.
[4] P. Graf, Substitution tree indexing, in: J. Hsiang (Ed.), Proceedings of the 6th International Conference on Rewriting Techniques and Applications (RTA-95), Lecture Notes in Computer Science, vol. 914, Kaiserslautern, Germany, 1995, pp. 117–131.
[5] T. Hillenbrand, B. Löchner, A phytography of waldmeister, AI Communications 15 (2–3) (2002) 127–133.
[6] N. Jones, C. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice-Hall, Englewood Cliffs, NJ, 1993.
[7] D. Kapur, P. Narendran, NP-completeness of the set unification and matching problems, in: 8th International Conference on Automated Deduction (CADE-8), Lecture Notes in Computer Science, vol. 230, Springer, Oxford, England, 1986, pp. 489–495.
[8] W.W. McCune, Experiments with discrimination-tree indexing and path indexing for term retrieval, Journal of Automated Reasoning 9 (2) (1992) 147–167.
[9] R. Nieuwenhuis, A. Rubio, Paramodulation-based theorem proving, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, vol. 1, Elsevier Science, Amsterdam, 2001, pp. 371–443 (Chapter 7).
[10] R. Nieuwenhuis, T. Hillenbrand, A. Riazanov, A. Voronkov, On the evaluation of indexing techniques for theorem proving, in: R. Goré, A. Leitsch, T. Nipkow (Eds.), First International Joint Conference on Automated Reasoning, IJCAR 2001, Lecture Notes in Artificial Intelligence, vol. 2083, Springer, Berlin, 2001, pp. 257–271, see also the COMPIT homepage www.mpi-sb.mpg.de/~hillen/compit/.
[11] W. Pugh, Skip lists: a probabilistic alternative to balanced trees, Communications of the ACM 33 (6) (1990) 668–676.
[12] R. Ramesh, I. Ramakrishnan, D. Warren, Automata-driven indexing of Prolog clauses, in: Seventh Annual ACM-Symposium on Principles of Programming Languages, San Francisco, 1990, pp. 281–291.
[13] A. Riazanov, A. Voronkov, The design and implementation of vampire, AI Communications 15 (2–3) (2002) 91–110.
[14] A. Riazanov, A. Voronkov, Partially adaptive code trees, in: M. Ojeda-Aciego, I. de Guzmán, G. Brewka, L. Pereira (Eds.), Logics in Artificial Intelligence. European Workshop, JELIA 2000, Lecture Notes in Artificial Intelligence, vol. 1919, Springer, Málaga, Spain, 2000, pp. 209–223.
[15] S. Schulz, E—a Brainiac Theorem Prover, AI Communications 15 (2–3) (2002) 111–126.
[16] R. Sekar, I. Ramakrishnan, A. Voronkov, Term indexing, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, vol. 2, Elsevier Science, Amsterdam, 2001, pp. 1853–1964 (Chapter 26).
[17] M. Stickel, The path indexing method for indexingterms, Technical Report 473, Artificial Intelligence Center, SRI International,Menlo Park, CA, October 1989.
[18] C.B. Suttner, G. Sutcliffe, The TPTP problemlibrary—v2.1.0, Tech. Rep. JCU-CS-97/8, Department of Computer Science, James CookUniversity (15 Dec. 1997). http://www.cs.jcu.edu.au/ftp/pub/techreports/97-8.ps.gz.
[19] A. Voronkov, The anatomy of vampire: implementing bottom-up procedures with code trees, Journal of Automated Reasoning 15 (2) (1995) 237–265.